

"Express Mail" Mailing Label No.: EV 304939233 US

Date of Deposit: August 29, 2003

Attorney Docket No. 13784US02

**SYSTEM AND METHOD FOR HANDLING OUT-OF-ORDER FRAMES**

**CROSS-REFERENCE TO RELATED APPLICATIONS/INCORPORATION BY  
REFERENCE**

**[01]** This application makes reference to, claims priority to, and claims benefit from:

United States Provisional Patent Application Serial No. 60/456,260, entitled "System and Method for Handling Out-of-Order Frames" filed on March 20, 2003;

United States Provisional Patent Application Serial No. 60/407,165, filed on August 30, 2002; and

United States Patent Application Serial No. \_\_\_\_\_ (Attorney Docket No. 15064US02), entitled "Method and System Data Placement of Out-Of-Order Frames" filed on August 29, 2003.

**[02]** The above stated applications are hereby incorporated herein by reference in their entirety.

**FIELD OF THE INVENTION**

**[03]** Certain embodiments of the invention relate generally to the processing of transmission packets, segments or frames. More specifically, certain embodiments of the invention relate to a method and system for handling out-of-order (OOO) frames.

**BACKGROUND OF THE INVENTION**

**[04]** Conventional transmission control protocol/internet protocol (TCP/IP) offload engines residing on network interface cards (NICs) or elsewhere in a system such

as in system software stacks, may inefficiently handle out-of-order (OOO) transmission control protocol (TCP) segments. For example, some conventional offload engines may merely drop out-of-order TCP segments. Dropped TCP segments need to be retransmitted by the sender, thereby utilizing additional bandwidth and reducing effective throughput. On links with large bandwidth-delay products such as high-speed local area network (LAN) of the order of 1 Gbps or faster, a large number of segments may be in transit between the sender and the receiver when the out-of-order TCP segment is dropped. Accordingly, many of the segments in transit must be retransmitted, thereby creating a substantial delay and excessive consumption of additional, often expensive and scarce bandwidth. TCP may also cut back on bandwidth allowed for a connection as the retransmission may be interpreted as being the result of congestion. This may further cause congestion avoidance mechanism to commence operation. A similar or even worse situation may arise with, for example, metropolitan area networks (MANs) with high bandwidth and moderate latencies or with long-haul wide area networks (WANs) that may have moderate bit rates and typical delays of the order of about 100 ms. In these types of networks, for example, system performance and throughput may be drastically reduced by the retransmissions.

**[05]** In some conventional systems, on the sender or transmitter side, TCPs generally begin transmission by injecting multiple TCP segments into the network corresponding to a maximum window size that may be indicated by a receiver. In networks in which traffic traverses multiple networking entities or devices having varying link speeds, some of the networking entities or devices may have to queue TCP segments in order to handle the traffic. For example, network devices such as routers especially interfacing faster links with slower links in the communication path between the transmitter side and the receiver side may have to queue TCP segments. In this regard, there may be instances when there is insufficient memory on the networking entities or devices for queuing the TCP segments resulting in

dropped segments. Accordingly, the TCP segments will have to be retransmitted, thereby consuming additional bandwidth.

[06] In certain systems, retransmission may trigger TCP slow start and congestion-avoidance procedures which may result in substantial decrease in available bandwidth of a communication link. TCP slow start is an algorithm that may be utilized to minimize the effects of lost packets that may result from insufficient memory on slower networking entities or devices. TCP slow start utilizes a congestion window that is initialized to one TCP segment at the time of link initiation. . In operation, the number of TCP segment allowed to be transmitted before an acknowledgment is received is incremented by one (1) for every acknowledgement (ACK) received from the remote peer. The sending side may therefore transmit a minimum number of TCP segments as specified by the minimum of the congestion window and the window that may be advertised by the receiving side. This may provide a near exponential growth in the window size and at some point, maximum capacity may be reached and the networking entity or device may start dropping packets.

[07] Congestion avoidance is an algorithm that may be utilized in conjunction with slow start to minimize the effects of lost packets. Congestion may occur when a device may receive more TCP segments at its input than it may be able to adequately process or more than it can send on the egress . Congestion may also occur when TCP segments transition from a faster transport infrastructure to a slower transport infrastructure. In this regard, the network device at the edge of the faster transport infrastructure and the slower transport infrastructure becomes a bottleneck. Congestion avoidance utilizes packet loss and duplicate acknowledgements (ACKs) to determine when congestion occurs. As a result, the sender rate may be cut by half every time congestion is experienced.

[08] Although slow start and congestion avoidance have varying objectives and are independent of each other, TCP recovery from congestion may involve decreasing the transmission rate and executing slow start to gradually increase the

transmission rate from a window size of one (1). In some cases, TCP on the remote peer generates numerous ACKs and the local peer's congestion avoidance may interpret this to mean that TCP segments are lost, resulting in retransmission. Accordingly, TCP recovery from congestion avoidance and/or TCP slow start can be a relatively slow process especially for high bandwidth and may in certain instances, also cause unwanted retransmissions.

[09] Other conventional offload engines may store out-of-order TCP segments in dedicated buffers attached to the offload engines residing on the NIC or a host memory until all the missing TCP segments have been received. The offload engine may then reorder and process the TCP segments. However, storing the TCP segments in dedicated buffers can be quite hardware intensive. For example, the size of the dedicated buffers scale with the product of the bandwidth of the connections times the delay on the connections, and with the number of connections. In addition, storing the out-of-order segments on dedicated buffers may consume precious processor bandwidth when the out-of-order segments have to be reordered and processed. In addition, the offload engine still needs to handle other segments arriving at wire speed. Therefore, the reordering and processing may have to occur at the expense of delaying the processing of currently received TCP segments or by over provisioning of processing power that is scarce and hard to acquire for high speed of networks.

[10] Accordingly, the computational power of the offload engine needs to be very high or at least the system needs a very large buffer to compensate for any additional delays due to the delayed processing of the out-of-order segments. When host memory is used for temporary storage of out-of-order segments, additional system memory bandwidth may be consumed when the previously out-of-order segments are copied to respective buffers. This choice complicates the processing of the data as the offload engine needs to communicate the state variables to a software agent for processing. While the software processes the state variables, the offload engine can't process new frames received for that TCP flow and has to buffer

them. When the software agent is done, it needs to move the state variables back to the offload engine. If on the other hand, the offload engine tries to process the data stored on the host memory instead of the software agent, it encounters longer latencies than when processing frames locally, making this option very low performance or almost impractical.

[11] Another design approach to a TCP offload Engine may be a flow-through approach. In the flow-through approach, the engine processes every TCP segment upon reception with no buffering, except for speed matching. The advantages of such a design approach are evidently the lack of external data buffering which scales with bandwidth delay product and with the number of connections. It adds cost, real estate and power to the solution as well additional pins on the offload engine ASIC to connect to the memory over a high speed bus. It also saves the additional complexity for reordering the out-of-order segments and processing them while additional traffic is received.

[12] However, one challenge generally faced by TCP implementers wishing to design a flow-through NIC, is that TCP segments may arrive out-of-order with respect to the order in which they were transmitted. This may prevent or otherwise hinder the immediate processing of TCP control data and prevent the placing of the data in a host buffer. Accordingly, an implementer may be faced with the option of dropping out-of-order TCP segments or storing the TCP segments locally on the NIC until all the missing segments have been received. Once all the TCP segments have been received, they may be reordered and processed accordingly. In instances where the TCP segments are dropped or otherwise discarded, the sending side may have to re-transmit all the dropped TCP segments and in some instances, may result in about a fifty percent (50%) or greater decrease in throughput or bandwidth utilization, as described above.

[13] Further limitations and disadvantages of conventional and traditional approaches will become apparent to one of skill in the art, through comparison of

such systems with some aspects of the present invention as set forth in the remainder of the present application with reference to the drawings.

## BRIEF SUMMARY OF THE INVENTION

[14] Some aspects of the present invention may be found in, for example, systems and methods that handle out-of-order frames. In one embodiment, the present invention may provide a method that handles out-of-order frames. The method may include, for example, one or more of the following: receiving an out-of-order frame via a network subsystem; placing data of the out-of-order frame in a host memory; and managing information relating to one or more holes in a receive window. In a further embodiment, the network subsystem may include, for example, a network controller or an offload engine. Moreover, the data of the out-of-order frame may be placed in a temporary buffer, an upper layer protocol (ULP) buffer or an application buffer residing in the host memory.

[15] In another embodiment, the present invention may provide a method that handles out-of-order frames. The method may include, for example, one or more of the following: parsing an out-of-order frame into control information and data information; processing at least one of the control information, the data information and context information to determine a buffer location in a host memory in which to place the data information; and managing receive window hole information. In a further embodiment, the receive window hole information may be, for example, TCP receive window hole information.

[16] In yet another embodiment, the present invention may provide a system that handles out-of-order frames. The system may include, for example, a host and a network subsystem. The host may include, for example, a host memory. The network subsystem may be coupled to the host. The network subsystem may process an out-of-order frame, place data of the out-of-order frame in the host memory, and manage information relating to one or more holes in a receive window. In a further embodiment, the network subsystem may include, for example, a network controller or an offload engine on a network interface card (NIC). In another further embodiment, the network subsystem may include, for example, a network

controller or an offload engine that is embedded on a motherboard or integrated into a main chipset.

[17] These and other advantages, aspects and novel features of the present invention, as well as details of a illustrated embodiment thereof, will be more fully understood from the following description and drawings.

## BRIEF DESCRIPTION OF THE DRAWINGS

[18] FIG. 1A is a block diagram of an exemplary system that may be utilized in connection with handling out-of-order frames in accordance with an embodiment of the invention.

[19] FIG. 1B is a block diagram of another exemplary system that may be utilized in connection with handling out-of-order frames in accordance with an embodiment of the invention.

[20] FIG. 1C is a block diagram of a system for handling out-of-order (OOO) transmission control protocol (TCP) datagrams in a flow-through manner in accordance with an embodiment of the invention.

[21] FIG. 1D is a flowchart illustrating exemplary steps for frame reception in accordance with an embodiment of the invention.

[22] FIG. 1E illustrates an exemplary transmission path in accordance with an embodiment of the invention.

[23] FIGS. 2A-C is a block diagram illustrating various out-of-order data inside a TCP receive window in connection with the handling out-of-order frames in accordance with an embodiment of the invention.

[24] FIGS. 3A-B is a block diagram illustrating exemplary variables that may be utilized to manage various out-of-order received frame scenarios in connection with handling out-of-order frames in accordance with an embodiment of the invention.

[25] FIGS. 4A-B is a block diagram illustrating an exemplary case where a new out-of-order TCP segment is received and is adjacent to existing hole in connection with handling out-of-order frames in accordance with an embodiment of the invention.

[26] FIGS. 5A-B is a block diagram illustrating an exemplary case where a new out-of-order TCP segment is received that plugs a first hole closest to the left edge

of a TCP window in connection with the handling out-of-order frames in accordance with an embodiment of the invention.

**[27]** FIGS. 6A-B is a diagram illustrating a case where a new out-of-order TCP segment is received that plugs another hole in connection with the handling out-of-order frames in accordance with an embodiment of the invention.

## DETAILED DESCRIPTION OF THE INVENTION

[28] FIG. 1A is a block diagram of an exemplary system that may be utilized in connection with handling out-of-order frames in accordance with an embodiment of the invention. Accordingly, the system of FIG. 1A may be adapted to handle out-of-order (OOO) transmission control protocol (TCP) datagrams in a flow-through manner according to aspects of the invention. Referring to FIG. 1A, the system may include, for example, a CPU 10, a memory controller 20, a host memory 30, a host interface 40, network subsystem 50 and an Ethernet 60. The network subsystem 50 may include, for example, a TCP-enabled Ethernet Controller (TEEC) or a TCP offload engine (TOE) 70. The network subsystem 50 may include, for example, a network interface card (NIC). The host interface 40 may be, for example, a peripheral component interconnect (PCI) or another type of bus. The memory controller 30 may be coupled to the CPU 20, to the memory 30 and to the host interface 40. The host interface 40 may be coupled to the network subsystem 50 via the TEEC or the TOE 70.

[29] FIG. 1B is a block diagram of another exemplary system that may be utilized in connection with handling out-of-order frames in accordance with an embodiment of the invention. Accordingly, the system of FIG. 1B may be adapted to handle out-of-order (OOO) transmission control protocol (TCP) datagrams in a flow-through manner according to aspects of the invention. Referring to FIG. 1B, the system may include, for example, the CPU 10, the host memory 30, a dedicated memory 35 and a chip set 55. The chip set 55 may include, for example, the network subsystem 50. The chip set 55 may be coupled to the CPU 10, to the host memory 30, to the dedicated memory 35 and to the Ethernet 60. The network subsystem 50 of the chip set 55 may be coupled to the Ethernet 60. The network subsystem 50 may include, for example, the TEEC or the TOE which may be coupled to the Ethernet 60. The dedicated memory 35 may provide buffers for context or data.

[30] Although illustrated, for example, as a CPU and an Ethernet, the present invention need not be so limited to such exemplary examples and may employ, for example, any type of processor and any type of data link layer or physical media, respectively. Accordingly, although illustrated as coupled to the Ethernet 60, the TEEC or the TOE 70 may be adapted for any type of data link layer or physical media. Furthermore, the present invention also contemplates different degrees of integration and separation between the components illustrated in FIGS. 1A-B. For example, the TEEC or the TOE 70 may be a separate integrated chip from the chip set 55 embedded on a motherboard or may be embedded in a NIC. In addition, the dedicated memory 35 may be integrated with the chip set 55 or may be integrated with the network subsystem 50.

[31] Some embodiments of the TEEC 70 are described in, for example, United States Patent Application Serial No. \_\_\_\_/\_\_\_\_,\_\_\_\_ (Attorney Docket No. 13782US02), entitled "System and Method for TCP Offload" filed on August 29, 2003. The above-referenced United States patent application is hereby incorporated herein by reference in its entirety.

[32] Some embodiments of the TOE 70 are described in, for example, United States Provisional Patent Application Serial No. 60/408,207, entitled "System and Method for Fault Tolerant TCP Offload" and filed on September 4, 2002; United States Patent Application Serial No. 10/337,029, entitled "System and Method for Fault Tolerant TCP Offload" and filed on January 6, 2003; United States Provisional Patent Application Serial No. 60/411,294, entitled "System and Method for Handling Partially Processed Frames" and filed on September 17, 2002; United States Patent Application Serial No. 10/302,474, entitled "System and Method for Handling Frames in Multiple Stack Environment" and filed on November 21, 2002; United States Provisional Patent Application Serial No. 60/410,022, entitled "System and Method for TCP Offloading and Uploading" and filed on September 11, 2002; United States Patent Application Serial No. 10/298,817, entitled "System and Method for TCP Offloading and Uploading" and filed on November 18, 2002; United States

Provisional Patent Application Serial No. 60/408,617, entitled "System and Method for TCP/IP Offload" and filed on September 6, 2002; and United States Patent Application Serial No. \_\_\_\_ (Attorney Docket No. 13782US02), entitled "System and Method for TCP/IP Offload" and filed on August 29, 2003. The above-referenced United States patent applications are all hereby incorporated herein by reference in their entirety.

[33] FIG. 1C is a block diagram of a system for handling out-of-order (OOO) transmission control protocol (TCP) datagrams in a flow-through manner in accordance with an embodiment of the invention. Referring to FIG. 1C, the incoming frame may be subject to L2 such as Ethernet processing including, for example, address filtering, frame validity and error detection. Unlike an ordinary Ethernet controller, the next stage of processing may include, for example, L3 such as IP processing and L4 such as TCP processing. The TEEC may reduce the host CPU utilization and memory bandwidth, for example, by processing traffic on hardware offloaded TCP/IP connections. The TEEC may detect, for example, the protocol to which incoming packets belong. If the protocol is TCP, then the TEEC may detect if the packet corresponds to an offloaded TCP connection, for example, a connection for which at least some TCP state information may be kept by the TEEC.

[34] Once a connection has been associated with a packet or frame, any higher level of processing such as L5 or above may be achieved. If the packet corresponds to an offloaded connection, then the TEEC may direct data movement of the data payload portion(s) of the frame. The destination of the payload data may be determined from the connection state information in combination with direction information within the frame. The destination may be a host memory, for example. Finally, the TEEC may update its internal TCP and higher levels of connection state and may obtain the host buffer address and length from its internal connection state.

[35] The receive system architecture may include, for example, a control path processing and data movement engine. The system components above the control

path as illustrated in upper portion of FIG. 1C, may be designed to deal with the various processing stages used to complete, for example, the L3/L4 or higher processing with maximal flexibility and efficiency and targeting wire speed. The result of the stages of processing may include, for example, one or more packet identification cards (PID\_Cs) that may provide a control structure that may carry information associated with the frame payload data. This may have been generated inside the TEEC while processing the packet in the various blocks. A data movement system as illustrated in the lower portion of FIG. 1C, may move the payload data portions of a frame along from, for example, an on-chip packet buffer and upon control processing completion, to a direct memory access (DMA) engine and subsequently to the host buffer that was chosen via processing.

[36] The receiving system may perform, for example, one or more of the following: parsing the TCP/IP headers; associating the frame with an end-to-end TCP/IP connection; fetching the TCP connection context; processing the TCP/IP headers; determining header/data boundaries; mapping the data to a host buffer(s); and transferring the data via a DMA engine into these buffer(s). The headers may be consumed on chip or transferred to the host via the DMA engine.

[37] The packet buffer may be an optional block in the receive system architecture of FIG. 1C. It may be utilized for the same purpose as, for example, a first-in-first-out (FIFO) data structure is utilized in a conventional L2 NIC or for storing higher layer traffic for additional processing.

[38] The packet buffer in the receive system of FIG. 1C may not be limited to a single instance. As control path processing is performed, the data path may store the data between data processing stages one or more times depending, for example, on protocol requirements.

[39] FIG. 1D is a flowchart illustrating exemplary steps for frame reception in accordance with an embodiment of the invention. Referring to FIG. 1C and FIG. 1D, in step 100, the NIC 50 may receive a frame from, for example, the Ethernet 60. In

step 110, the frame parser may parse the frame, for example, to find the L3 and L4 headers. The frame parser may process the L2 headers leading up to the L3 header, for example IP version 4 (IPv4) header or IP version 6 (IPv6) header. The IP header version field may determine whether the frame carries an IPv4 datagram or an IPv6 datagram.

[40] For example, if the IP header version field carries a value of 4, then the frame may carry an IPv4 datagram. If, for example, the IP header version field carries a value of 6, then the frame may carry an IPv6 datagram. The IP header fields may be extracted, thereby obtaining, for example, the IP source (IP SRC) address, the IP destination (IP DST) address, and the IPv4 header “Protocol” field or the IPv6 “Next Header”. If the IPv4 “Protocol” header field or the IPv6 “Next Header” header field carries a value of 6, then the following header may be a TCP header. The results of the parsing are added to the PID\_C and the PID\_C travels with the packet inside the TEEC.

[41] The rest of the IP processing may subsequently take place later in a manner similar to the processing in a conventional off-the-shelf software stack. Implementation may vary from the use of firmware on an embedded processor to a dedicated, finite state machine, which may be potentially faster, or a hybrid of a processor and a state machine. The implementation may vary with, for example, multiple stages of processing by one or more processors, state machines, or hybrids. The IP processing may include, for example, extracting information relating to, for example, length, validity, fragmentation, etc. The located TCP header may also be parsed and processed. The parsing of the TCP header may extract information relating to, for example, the source port and the destination port.

[42] The TCP processing may be divided into a plurality of additional processing stages. In step 120, the frame may be associated with an end-to-end TCP/IP connection. After L2 processing, in one embodiment, the present invention may provide that the TCP checksum be verified. The end-to-end connection may be uniquely defined by, for example, the following 5 tuple: IP Source address (IP SRC

addr); IP destination address (IP DST addr); L4 protocol above the IP protocol such as TCP, UDP or other upper layer protocol; TCP source port number (TCP SRC); and TCP destination port number (TCP DST). The process may be applicable for IPv4 or IPv6 with the choice of the relevant IP address.

[43] As a result of the frame parsing in step 110, the 5 tuple may be completely extracted and may be available inside the PID\_C. Association hardware may compare the received 5 tuple with a list of 5 tuples stored in the TEEC. The TEEC may maintain a list of tuples representing, for example, previously handled off-loaded connections or off-loaded connections being managed by the TEEC. The memory resources used for storing the association information may be costly for on-chip and off-chip options. Therefore, it is possible that not all of the association information may be housed on chip. A cache may be used to store the most active connections on chip. If a match is found, then the TEEC may be managing the particular TCP/IP connection with the matching 5 tuple.

[44] FIG. 1E illustrates an exemplary transmission path in accordance with an embodiment of the invention. Referring to FIG. 1E, the TEEC may include, for example, a physical layer (PHY) 180, a MAC layer 190, a header builder 200, a context prefetch 210, a timer 220, a transmit processor, with for example, transmit code 230, data and control blocks 240, acknowledge blocks 250, a scheduler 260 and a DMA engine 270. The timer 220 may include, for example, retransmit timers. The scheduler 260 may be adapted for functions, including but not limited to, allocating bandwidth to connections that have data to be transmitted, based on remote availability of TCP buffers determined by TCP window and/or retransmit arbitration. The DMA engine 270 may include, for example, an XSUM block 280 or other data specific processing. This may include insertion of data into the data provided by the host and computing CRC values. The data processing is not limited to these functions.

[45] On the transmission path, the support for L4 and higher levels may include additional complexity and functionality. Transmission may include performing, for

example, one or more of the following: scheduling transmission flow; transferring data via DMA; fetching context; transmit processing; adding L5 or higher and TCP/IP headers; arming the timers; and L2 transmitting.

[46] The scheduler 260 may decide which flow to serve next. The scheduler 260 may also handle multiplexing L2 and L4 and higher levels of traffic. With respect to L4 and higher levels of traffic, the decision to schedule a particular TCP/IP flow for transmission may depend upon, for example, one or more of the following factors: data availability for transmission on the host side; remote end buffering state such as when a remote TCP connection has not closed the TCP window; preventing starvation among the potentially large number of TCP connections competing for time on the Ethernet media; availability of TCP acknowledges from the receive side; and a need to retransmit information on behalf of a TCP connection.

[47] Utilizing some or all of the above-identified information or other information, the scheduler 260 may pick the next flow to be transmitted. The scheduler 260 may fetch the pointer to the next host resident buffer from the context information. The scheduler 260 may also program the DMA engine 270 to get the data and store the data, for example, in the on-chip FIFO buffer.

[48] The DMA engine 270 may transfer the data from the host buffer or buffers into, for example, an on-chip, transmit-side FIFO buffer. IP Checksum (IPv4) and TCP Checksum may be computed on the data being transferred. The computations may be performed concurrently with data movement. Higher levels of data processing may be done at this stage.

[49] Flow context may be fetched, for example, by the context prefetch 220 from a central context resource. Accessing the central context resource may increase the usefulness of a locking mechanism among all of its consumers to ensure data integrity and coherency. The locking mechanism may be very efficient in minimizing undesired performance impact. The context for the selected flow may be provided to the transmit processor 230 such as a CPU and/or finite state machine (FSM).

[50] The transmit processor 230 or finite state machine or a combination thereof, may be adapted, for example, to execute TCP/IP and higher levels of code, to update the context and to generate the TCP/IP and higher level header variables to be placed in the headers. The updated context may be stored. The processing for this stage may be performed by one or more stages including one or more processors, state machines or hybrid processors.

[51] The header builder 200 may use the header variables generated by transmit processor 230 and may generate the TCP/IP and higher level headers and may attach the TCP/IP and higher level headers in front of the data to be transmitted. Using the partial checksum results obtained from the DMA engine 270, the header builder 200 may finalize the checksum fields and may place them in the respective headers. Transmit processing is not limited to a specific number of stages and processing and may be executed at different stages of processing as may be optimal.

[52] The timer 220 may be armed by the transmit processor 230 and may update its future timer event list. When the L4 and higher levels of processing are complete, the L2 processing and transmission may follow the conventional steps performed by a conventional Ethernet controller.

[53] A retransmission event is similar to a normal transmission except, for example, the data to be retransmitted may be fetched from the host buffers as before or from any other temporary holding buffers. The address for this data may be computed. The computation of the host buffer address may be more complex. The same mapping function described in the receive path section may be utilized for the retransmission TCP sequence number range. Once the buffer address is determined, the rest of the transmit process may occur as described above.

[54] Exemplary receive and transmit paths a TEEC in accordance with an embodiment of the invention are described in United States Patent Application Serial

No. \_\_\_\_\_ (Attorney Docket No. 13782US02) filed August 29, 2003 and is incorporated herein by reference in its entirety.

**[55]** Hereinafter, although reference shall be made with respect to the TEEC 70, the present invention also contemplates using a TOE 70 instead of or in combination with the TEEC 70. Accordingly, descriptions relating to the TEEC 70 may also be applied with respect to the TOE 70. The TEEC is an engine that performs, among other things, TCP/IP processing in a flow through manner.

**[56]** In operation, on the receive path, a frame may be received by the network subsystem 50 from the Ethernet 60 in accordance with an embodiment of the present invention. The network subsystem 50 may include, for example, a NIC. The TEEC 70, for example, may parse and may process the headers of the received frame. The control information may be parsed from the data information of the received frame. Parsing may include, for example, one or more of the following: finding boundaries of fields with control information, checking for correctness and separating control information from data information. The TEEC 70 may process at least one of the control information, the data information and context information (e.g., state information about the connection with which the out-of-order frame may be associated) in determining, for example, whether the received frame is a valid frame. In one embodiment, a valid frame may be defined as a frame within the TCP window (e.g., a receive window). In another embodiment, an invalid frame may be defined as a frame not within the TCP window or as a duplicate frame in the TCP window. If the frame is not valid, then the frame may be dropped. If the frame is valid, then the TEEC 70 may process at least one of, for example, the control information, the data information and the context information to place the data information into, for example, a particular temporary buffer, an upper layer protocol (ULP) buffer or an application buffer in the memory 30 of the host via the host interface 40 and the memory controller 20. In one embodiment, if only a portion of the frame is valid, then only the valid portion of the data information of the frame may be placed into the host memory 30.

[57] The placement of data information may be performed, for example, by mapping the TCP space into the buffer space. The buffer space may be represented as, for example, a linear space. The first byte of the buffer may correspond to a particular TCP sequence value. Other bytes in the TCP segment may be placed via offsets in the buffer that may correspond to respective deltas in the TCP sequence space with respect to the sequence value of the first byte. Some embodiments may provide for a ULP-specific translation that maps particular ranges of the TCP sequence numbers to particular buffer areas.

[58] In one embodiment, the TEEC 70 may associate the incoming frame with, for example, an established connection that is being managed by the TEEC 70. The placement of the data information of the frame in the TCP window, for example, may create or may modify one or more holes in the TCP window. Information about any of the existing holes in the TCP window may be stored and managed (e.g., generated, updated, copied, deleted, etc.) in the TEEC 70 or stored in a memory (e.g., a dedicated memory) that may be on the network subsystem 50 (e.g., on a NIC), off the network subsystem 50 (e.g., off a NIC), on the chip set 55, off the chip set 55, embedded on a motherboard or some combination thereof.

[59] Some embodiments of the present invention may provide for a selectable (e.g., programmable) limit on the number of holes that the TEEC 70 may manage within the TCP window. The selectable limit may represent, for example, an optimal number of holes that a flow-through network subsystem 50 (e.g., a flow-through NIC of the network subsystem 50) may support. The optimal number of holes may depend upon, for example, the type of connection. For example, a wide area network (WAN) connection may drop frames mainly due to router congestion yielding potentially many holes in a large window. In another example, a local area network (LAN) connection may have a single drop due to a switch drop before a link-by-link flow control is triggered thereby preventing more traffic from flowing.

[60] Some embodiments of the present invention may provide that the network subsystem 50 support a ULP on top of TCP. The control information and the data

information may be separated from incoming frames with the data information being placed in the host memory 30 and possibly buffering some control information on the network subsystem 50. In some cases, more state information may have to be maintained or managed per hole. Furthermore, ULP-specific activities may be employed once a hole is plugged or once the receiver receives more information about the behavior expected by the ULP in the control plane (e.g., to notify the ULP of a completion of a message).

**[61]** FIGS. 2A-C is a block diagram illustrating various out-of-order data inside a TCP receive window in connection with the handling out-of-order frames in accordance with an embodiment of the invention. In FIG. 2A, a TCP window may be defined in TCP sequence space. In one embodiment, the TCP window may have a left boundary at a TCP sequence value of RCV\_NXT and a right boundary at a TCP sequence value of RCV\_NXT + RCV\_WIND. RCV\_NXT may be a variable, for example, used to keep track of the next expected sequence number to be received by a receiver. RCV\_WIND may be a variable, for example, used to determine the width of the TCP window in TCP sequence space. Although not illustrated, some embodiments of the present invention contemplate that TCP space wraps around after a particular TCP sequence number. For example, the TCP space may wrap around every  $2^{32} - 1$ .

**[62]** In FIG. 2B, an out-of-order frame may be received by the network subsystem 50. The TEEC 70 may parse and may process the out-of-order frame. In one embodiment, control information may be separated from the data information. The control information may be stored, for example, on the network subsystem 50 or, in particular, in the TEEC 70. The network subsystem 50 may include, for example, a dedicated memory on which control information related to out-of-order frames may be stored. Context information may also be stored in the dedicated memory. In one example, the dedicated memory may be off the network subsystem 50 or distributed between off-network-subsystem sites (e.g., off-NIC sites) and on-network-subsystem sites (e.g., on-NIC sites). The control information may include, for example, hole

information. The TEEC 70 may process at least one of the control information, the data information and context information (e.g., state information about the connection with which the out-of-order frame may be associated) to determine a location in the host memory 30 in which to place the data information.

**[63]** In one embodiment, the data information may be placed in, for example, a temporary buffer, an upper layer protocol (ULP) buffer or an application buffer residing in the host memory 30. The reception of the out-of-order data information may be referred to as a hole in the TCP window. The first hole may be defined by at least two variables: Hole\_1\_Start and Hole\_1\_End. Hole\_1\_Start may be defined, for example, as the TCP sequence value of the beginning of the first hole in TCP sequence space. Hole\_1\_End may be defined, for example, as the TCP sequence value of the ending of the first hole in TCP sequence space. The variables describing the first hole may be maintained, for example, as control information in the TEEC 70, on the network subsystem 50, off the network subsystem 50 or some combination thereof as described above. In one embodiment, although the data information of the out-of-order frame may be stored in the host memory 30 in a flow-through manner, the control information relating to the out-of-order frame may be stored on the network subsystem 50, for example, as long as the information is useful to the network subsystem 50 in processing incoming frames. The control information relating to the out-of-order frame may be stored on the network subsystem 50, for example, in a summary table or other data structure.

**[64]** In FIG. 2C, a second out-of-order frame may be received by the network subsystem 50. As with the first out-of-order frame, in a flow-through process, the data information of the second out-of-order frame may be placed into the host memory 30. As illustrated in FIG. 2C, the data information of the second out-of-order frame may create a second hole. Accordingly, information about the second hole including, for example, Hole\_2\_Start and Hole\_2\_End, may be maintained as part of the control information relating to the second out-of-order frame. As with the first out-of-order frame, although the data information is placed into the host memory

30, the control information relating to the second out-of-order frame may be stored in the TEEC 70 or in a dedicated memory that may be on the network subsystem 50, off the network subsystem 50 or some combination thereof.

**[65]** FIGS. 3A-B is a block diagram illustrating exemplary variables that may be utilized to manage various out-of-order received frame scenarios in connection with handling out-of-order frames in accordance with an embodiment of the invention. FIG. 3A shows that the TCP window already has two holes which the network subsystem 50 via, for example, the TEEC 70 is managing. In FIG. 3B, an in-order frame may be received by the network subsystem 50. The TEEC 70 may parse and may process the in-order frame. In one embodiment, control information may be separated from the data information. The TEEC 70 may process at least one of, for example, the control information, the data information and the context information to determine a location in the host memory 30 in which to place the data information. As illustrated in FIG. 3B, the placement of the data information from the in-order frame may modify the first hole and the TCP window. The first hole may be reduced in size and one or more the first hole variables may be updated. In the illustrated embodiment, the value of Hole\_1\_Start is increased and the value of Hole\_1\_End remains the same. The left edge of the TCP window may be shifted to the right by adjusting the value of RCV\_NXT to accommodate the addition of the data information of the in-order frame to the front portion of the previous TCP window. In one example, the value of RCV\_NXT may be increased by the value of the width in TCP sequence space of the newly placed data information of the in-order frame. In another example, the TCP window may shrink in width by the width in TCP sequence space of the newly placed data information. In yet another example, the receiver may have additional resources (e.g., buffers) and may advertise them to its remote peer, thus the TCP window may not change in width or may grow in width.

**[66]** FIGS. 4A-B is a block diagram illustrating an exemplary case where a new out-of-order TCP segment is received and is adjacent to existing hole in connection with handling out-of-order frames in accordance with an embodiment of the

invention. FIG. 4A shows that the TCP window already has two holes which the network subsystem 50 via, for example, the TEEC 70 is managing. In FIG. 4B, an out-of-order frame may be received by the network subsystem 50. As illustrated in FIG. 4B, the placement of the data information from the out-of-order frame may modify, for example, the first hole. The first hole may be reduced in size and one or more the first hole variables may be updated. In the illustrated embodiment, the value of Hole\_1\_Start remains the same and the value of Hole\_1\_End decreases. The TCP window may remain the same.

[67] FIGS. 5A-B is a block diagram illustrating an exemplary case where a new out-of-order TCP segment is received that plugs a first hole closest to the left edge of a TCP window in connection with the handling out-of-order frames in accordance with an embodiment of the invention. FIG. 5A shows that the TCP window already has two holes which the network subsystem 50 via, for example, the TEEC 70 is managing. In FIG. 5B, an in-order frame may be received by the network subsystem 50. The placement of the data information from the in-order frame may completely plug the first hole and modify the TCP window. In the illustrated embodiment, the former first hole is completely plugged. Accordingly, the network subsystem 50 need only manage the one remaining hole. In one embodiment, the first hole variables may be deleted and the second hole variables may become the new first hole variables and the second hole variable values may become the new first hole variable values. If the TCP window included  $m$  holes, in which  $m$  is an integral value, and the first hole were plugged, then  $n$ -th hole variables, in which  $n$  is an integral value less than or equal to  $m$ , may become the new  $(n-1)$ -th hole variables and the  $n$ -th hole variable values may become the new  $(n-1)$ -th hole variable values. The left edge of the TCP window may be shifted to the right by adjusting the value of RCV\_NXT to accommodate the addition of the data information of the in-order frame and the previously placed out-of-order frame(s) to the front portion of the former TCP window. In one example, the value of RCV\_NXT may be increased by the value of the width in TCP sequence space of the newly placed data information of the in-

order frame and the width in TCP sequence space of the previously placed out-of-order frame(s) between the former first hole and the former second hole. As described above, the TCP window may shrink in width, increase in width or remain the same width.

**[68]** FIGS. 6A-B is a diagram illustrating a case where a new out-of-order TCP segment is received that plugs another hole in connection with the handling out-of-order frames in accordance with an embodiment of the invention. FIG. 6A shows that the TCP window already has two holes which the network subsystem 50 via, for example, the TEEC 70 is managing. In FIG. 6B, an out-of-order frame may be received by the network subsystem 50. The placement of the data information from the out-of-order frame may completely plug the second hole. Accordingly, the network subsystem 50 need only manage the one remaining hole. In one embodiment, the second hole variables may be deleted. The TCP window may remain the same.

**[69]** The pseudo code, as set forth below, describes an embodiment of a process that moves data from the TCP segment to the buffer list. For simplicity the code that may cover handling the PUSH bit or that may cover the “no room in current buffer list” has been omitted. For further simplicity, the TCP situation is assumed to be static such that there may be no receive TCP window update. Another simplification is that the wrap around of the TCP sequence number space may not be addressed in the pseudo code.

```
/* global initializations */
Number_of_holes = 0;
OOO_Data {
    Begin Int32;
    End Int32;
    ULP_Event_present Boolean;           /* Variable keeps tag of
                                             presence of ULP event in OOO segment */
```

```

    ULP_Event_TCP_Sequence_number Int32; /* TCP sequence number
for ULP event */

} OOO_list[l]; /* this list of structure keeps tag of TCP seq number of
segments received o-o-o */

/* RCV_NXT is the TCP sequence number of next expected byte */

1. /* The TCP Sequence number range (TCP Seq # of the first byte
of TCP payload is First_TCP_Seq, TCP Sequence # of last byte is
Last_TCP_Seq) is checked to be within RCV window (between
RCV_NXT and RCV_NXT + TCP_Window) */

1A. /* If frame outside the window (stale or duplicate) drop the frame
*/
    if ((First_TCP_Seq > (RCV_NXT + TCP_window)) || (Last_TCP_Seq <
RCV_NXT)) then drop_frame();

1B. /* If some bytes have been received before, ignore duplicate bytes */
    if First_TCP_Seq < RCV_NXT then First_TCP_Seq = RCV_NXT;

1C. /* In case some bytes are out of the window, the receiver may
drop the whole segment. Here the receiver drops the illegal bytes only */
    if Last_TCP_Seq > (RCV_NXT + TCP_window) then
Last_TCP_Seq = (RCV_NXT + TCP_window);

1D. /* In case last data is o-o-o; do accounting. It may be a new hole or
plug a hole or be adjacent to a hole */
    if First_TCP_Seq > RCV_NXT then Number_of_holes =
Hole_accounting();

1E. /* In case last data is in-order, it may plug a hole */
    if (First_TCP_Seq == RCV_NXT) RCV_NXT = Last_TCP_Seq + 1; /* update of the TCP_Window is outside the scope of this code */
    {
        if ((Number_of_Holes > 0) && (Last_TCP_Seq +1 ==
OOO_list[1].Begin)) then
        {
            /* removes element 1, index of following elements reduced
by one */
            Remove_OOO_list_element(1);

```

```

        Number_of_Holes--;
    }
}

2.      /* Find the right entry in the buffer list. Placement is oblivious to
the in-order or part of a hole issue */

    Segment_Length = Last_TCP_Seq - First_TCP_Seq;

/* Delta holds the difference in the TCP sequence number to the location
first byte of the frame. It is also the distance in the buffer space to the first
byte that ought to used to store it */

    Delta = (First_TCP_Seq - RCV_NXT) /* delta from anchor */

    i=0;                  /* used to dynamically point to the buffer that
corresponds to RCV_NXT */

/* Some bytes of Host_Buffer_List.Buffer[0] may have been used already.
Need to figure out how many are left */

    if (Delta < (Host_Buffer_List.Buffer[0].length -
(Host_Buffer_List.Buffer[0].NXT_Addr -
Host_Buffer_List.Buffer[0].Phy_Addr)))

    {
        Delta += (Host_Buffer_List.Buffer[0].NXT_Addr -
Host_Buffer_List.Buffer[0].Phy_Addr);
    }
    else
    {
        Delta -= (Host_Buffer_List.Buffer[0].length -
(Host_Buffer_List.Buffer[0].NXT_Addr -
Host_Buffer_List.Buffer[0].Phy_Addr));
    }

5.      Do while { Delta - (Host_Buffer_List.Buffer[i].length > 0}

{
    Delta -= Host_Buffer_List.Buffer[i].length;
    i++;
}
i--;
}

```

```

6.      /* The variable i point to the l-th buffer after the first buffer, where
data posting should start at. Delta holds the offset into this buffer */
7.      Bytes_to_DMA = Segment_length;
8.      /* DMA into first buffer, syntax of DMA_Data (from address, to
address, length) */
DMA_Data (First_TCP_Seq, Host_Buffer_List.Buffer[l].Phy_Address+
Delta, Host_Buffer_List.Buffer[l].length-Delta)

10. /* is_buffer_full has the following syntax (first_byte_written, length)
and returns 1 in case it is full */
if (buff_full = is_buffer-Full(Host_Buffer_List.Buffer[l].Phy_Address+
Delta, Host_Buffer_List.Buffer[l].length-Delta) ) then
return_buffer_to_owner();

11. Bytes_to_DMA -= Host_Buffer_List.Buffer[l].length- Delta; /*  

bytes that have been DMA'd into 1st buffer */
12. Start_TCP_Seq = First_TCP_Seq +
(Host_Buffer_List.Buffer[l].length - Delta);
/* DMA into next buffers if needed */
13. Do while { Bytes_to_DMA > 0}
14. {
if (Bytes_to_DMA > Host_Buffer_List.Buffer[l].Length) DMA data
(Start_TCP_Seq, Host_Buffer_List.Buffer[l].Phy_Address,
Host_Buffer_List.Buffer[l].Length)
else DMA data (Start_TCP_Seq,
Host_Buffer_List.Buffer[l].Phy_Address, Bytes_to_DMA);

Bytes_to_DMA -= Host_Buffer_List.Buffer[l].length;
Start_TCP_Seq += Host_Buffer_List.Buffer[l].length
l++;
If l > max_buffers then goto no_more_buffers;
}

}

```

```

/* Routine for accounting for in-window TCP data that is received out-of-
order. Builds o-o-o data management structures, returns number of holes
*/
int32
Hole_accounting()
{
    /* check no duplicate of o-o-o data */
    l = 0;
    Do while ( Number_of_Holes > l )
    {
        /* For simplicity of the pseudo code the case where one TCP segment
        overlaps more than one hole is not discussed here. However it is a simple
        extrapolation of the code below */
        If ((First_TCP_Seq < (OOO_list[l].Begin) && (Last_TCP_Seq >
        (OOO_list[l].Begin) && (Last_TCP_Seq <= (OOO_list[l].End)) then
        Last_TCP_seq = (OOO_list[l].Begin);

        If ((First_TCP_Seq >= (OOO_list[l].Begin) && (Last_TCP_Seq <=
        (OOO_list[l].End)) then drop_frame();

        If ((First_TCP_Seq >= (OOO_list[l].Begin) && (First_TCP_Seq <
        (OOO_list[l].End) && (Last_TCP_Seq > (OOO_list[l].End) then
        First_TCP_seq = (OOO_list[l].End);

        /* Is this data adjacent to existing hole? Move edge of hole. Is the hole
        plugged now? */
        If ((First_TCP_Seq == OOO_list[l].End+1) then
        {
            OOO_list[l].End = Last_TCP_Seq;
            If (OOO_list[l].End +1 == OOO_list[l+1].Begin) then
            {
                OOO_list[l].End = OOO_list[l+1].End;
                /* removes element l+1, index of following elements reduced by
                one */
                Remove_OOO_list_element(l+1);
            }
        }
    }
}

```

```

        return (Number_of_Holes--);
    }
}

If (Last_TCP_Seq+1 == OOO_list[l].Begin) then
    OOO_list[l].Begin = First_TCP_Seq;
If (OOO_list[l].Begin == OOO_list[l-1].End +1) then
{
    OOO_list[l-1].End = OOO_list[l].End;
/* removes element l, index of following elements reduced by one
*/
    Remove_OOO_list_element(l);
    return (Number_of_Holes--);
}
}

/* if new hole allocate space to manage it */
If ((First_TCP_Seq > OOO_list[l].End+1) && (First_TCP_Seq+1
< OOO_list[l+1].Begin) then
{
    /* insert element l+1 */
    Insert_OOO_list_element(l+1);
    OOO_list[l+1].Begin = First_TCP_Seq;
    OOO_list[l+1].End = Last_TCP_Seq;
    return (Number_of_Holes++);
}
}
}

```

[70] The base sequence number and the host buffer info list may be updated as buffers are consumed along with the movement of the TCP Window to the right.

[71] By accounting for missing TCP segments (e.g., holes), the data information of incoming frames may be placed in a flow-through manner into a host buffer. The data information need not be locally stored on the network subsystem 50 nor held until other TCP segments are received thereby completing an in-order set of TCP segments. This may be accomplished, for example, by splitting the control data from the data information. The data information may be stored in the host buffer and a summary of control data may be stored on, for example, the network subsystem 50. Since the network subsystem 50 is not required to support a buffer for out-of-order TCP segments, the network subsystem 50 may use a resource-efficient scalable memory to manage the control information relating to the out-of-order TCP segments. The memory for data on the network subsystem 50 may then scale with the number of holes and not with the number of frames that are out of order. Indeed, in some embodiments, the memory for data need not scale with the number of connections, the number of holes per connection or even the TCP connection, bandwidth-delay product.

[72] Accordingly, the present invention may be realized in hardware, software, or a combination of hardware and software. The present invention may be realized in a centralized fashion in one computer system, or in a distributed fashion where different elements are spread across several interconnected computer systems. Any kind of computer system or other apparatus adapted for carrying out the methods described herein may be suited. A typical combination of hardware and software may be a general-purpose computer system with a computer program that, when being loaded and executed, controls the computer system such that it carries out the methods described herein.

[73] The present invention may also be embedded in a computer program product, which comprises all the features enabling the implementation of the methods described herein, and which when loaded in a computer system is able to carry out these methods. Computer program in the present context means any expression, in any language, code or notation, of a set of instructions intended to

cause a system having an information processing capability to perform a particular function either directly or after either or both of the following: a) conversion to another language, code or notation; b) reproduction in a different material form.

**[74]** While the present invention has been described with reference to certain embodiments, it will be understood by those skilled in the art that various changes may be made and equivalents may be substituted without departing from the scope of the present invention. In addition, many modifications may be made to adapt a particular situation or material to the teachings of the present invention without departing from its scope. Therefore, it is intended that the present invention not be limited to the particular embodiment disclosed, but that the present invention will include all embodiments falling within the scope of the appended claims.